Usability Testing of a Graphical Programming System Things We Missed in a Programming Walkthrough

Brigham Bell
John Rieman
Clayton Lewis

CU-CS-498-90 October 1990

* Supported by NSF NYI #CCR-9357740, ONR #N00014-96-1-0720, and a Packard Fellowship in Science and Engineering from the David and Lucile Packard Foundation.

Report Documentation Page					*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **OCT 1990**	2. REPORT TYPE	3. DATES COVERED **00-00-1990 to 00-00-1990**
4. TITLE AND SUBTITLE **Usability Testing of a Graphical Programming System Things We Missed in a Programming Walkthrough**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Colorado at Boulder,Department of Computer Science,Boulder,Co,80309**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR’S ACRONYM(S)
		11. SPONSOR/MONITOR’S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**13**	

Usability Testing of a Graphical Programming System: Things We Missed in a Programming Walkthrough

Brigham Bell[1,2]
John Rieman[1,2]
Clayton Lewis[1,2,3]

CU-CS-498-90 October 1990

[1]Institute of Cognitive Science
[2]Department of Computer Science
[3]Center for Advanced Decision Support for Water and Environmental Systems

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309
303-492-4932
bell@boulder.colorado.edu

This work was supported in part by grant no. IRI-8944178 from the National Science Foundation. Related research was supported by US West Advanced Technologies.

Usability Testing of a Graphical Programming System: Things We Missed in a Programming Walkthrough

Brigham Bell[1,2], *John Rieman*[1,2], and *Clayton Lewis*[1,2,3]

[1]Institute of Cognitive Science
[2]Department of Computer Science
[3]Center for Advanced Decision Support for Water and Environmental Systems

University of Colorado
Boulder, Colorado 80309

ABSTRACT
Traditional programming language design has focussed on efficiency and expressiveness, with minimal attention to the ease with which a programmer can translate task requirements into statements in the language, a characteristic we call "facility." The programming walkthrough is a method for assessing the facility of a language design before implementation. We describe the method and its predictions for a graphical programming language, ChemTrains. These predictions are contrasted with protocols of subjects attempting to write their first ChemTrains program. We conclude that the walkthrough is a valuable aid at the design stage, but it is not infallible. Our results also suggest that it may not be enough for programmers to know how to solve a problem; they must also understand why the solution will succeed.

KEYWORDS: language design, graphical programming, usability evaluation, walkthrough.

I. INTRODUCTION

The design, implementation, and testing of a new programming language is a time-consuming and labor-intensive process. Much attention has been given to the features that a language needs to ensure its efficiency and expressive power [11], and these qualities can be analyzed to some extent at the design stage. Less effort has been directed at improving a language's facility, the ease with which a programmer can translate the requirements of a task into statements in the language. The work of Brooks [3], Soloway and colleagues [17,18], and Green and colleagues [6,16] has produced some understanding of the mental processes that underlie programming, but language designers have generally not used this work to guide their efforts.

We have developed a procedure called the *programming walkthrough*, which evaluates a language's ease of use at the design stage. The procedure is an analysis at the knowledge-level [13] which identifies the language-specific facts needed to perform one or more well-defined tasks with a proposed language design. In this paper we summarize the results of a programming walkthrough that evaluated the facility of ChemTrains, a graphical language intended to allow nonprogrammers to create animated simulations. We then describe empirical testing in which subjects were asked to write a program in a prototype ChemTrains implementation. We compare the predictions of the walkthrough to the subjects' behavior in the tests, and conclude that the walkthrough successfully predicts much of the required knowledge. We also find, however, that programmers' metacognitive strategies sometimes demand more knowledge than a pure knowledge-level analysis would suggest.

II. USING THE PROGRAMMING WALKTHROUGH METHOD TO DESIGN CHEMTRAINS

In this section we describe the programming walkthrough methodology, and how the method was applied during the design of the ChemTrains language.

The Programming Walkthrough -- A Knowledge-Level Evaluation

The programming walkthrough is a method of evaluating two important aspects of a programming environment: *expressiveness*, the ability to state solutions to hard problems simply, and *facility*, the ability to solve problems easily. In the walkthrough, a programming environment is evaluated by analyzing the mental steps that a programmer would most likely take in solving a specific problem. The method is an outgrowth of work on cognitive walkthroughs, which are used to critique end-user interfaces [9].

The programming walkthrough method requires two things: a representative set of tasks or problems that the system is intended for, and a document describing what a naive user needs to know about the system, which we call the *doctrine*. Doctrine includes general concepts of the system and its use, as well as advice on how to go about solving problems. The problem-solving advice supplies what Soloway calls *plans,* which the programmer needs to bridge between task requirements and individual programming statements or operations [18].

Once the doctrine is fixed, the facility of a design is evaluated by doing walkthroughs on sample problems. A sequence of decisions adequate to develop a solution to each problem is outlined, and the difficulty of each decision is assessed. If all decisions are directly guided by doctrine, facility will be good. If there are decisions that are unguided, or for which wrong choices are consistent with doctrine, facility will be poor. The nature and quantity of doctrine is also considered: a design that requires little doctrine, and doctrine that is judged easy to understand, should show greater facility than one which needs much complex doctrine. We used this way of assessing facility to guide the design of the ChemTrains graphical programming system.

Design of ChemTrains

ChemTrains is intended to permit users without programming background to construct animated graphical models of systems for which they have a qualitative behavior model, such as document flow in an organization or the phase change of a substance as temperature varies. ChemTrains models show objects participating in reactions similar to chemical reactions and moving among places on the screen along paths. The name "ChemTrains" was suggested by the role of the reactions and the role of paths, thought of as railroad tracks.

The design process we used for ChemTrains is described at length in Lewis, et al. [10]. After a design period in which six problems were used as a platform to discuss design alternatives, we produced and recorded a design space in which decisions on 28 specific design features could determine a unique ChemTrains design. Here are some of the features, which had been stated as yes/no questions:

- Do objects have text labels associated with them?
- Can places overlap?
- Do reaction rules support the use of variables?
- Are reaction rules specified by demonstration?
- Can reactions occur between objects in multiple places on the screen?

At this point we stopped exploring new design features and defined three complete designs, ZeroTrains, ShowTrains, and OPSTrains, which represented distinct combinations of features within the design space. In ZeroTrains the simpler alternative for a feature was chosen whenever possible. OPSTrains chose features for power and was influenced by the OPS family of rule-based systems [5]. ShowTrains chose a mix of powerful features, but mainly differs from the other designs by how the rules are specified. In ShowTrains rules are expressed by demonstration, a technique suggested by Maulsby's work on specifying procedures by example [12].

Walkthrough Evaluation of Alternative Designs

Doctrine, the language-specific knowledge required to do the programming walkthroughs, was developed iteratively, using the following procedure:

- Prepare an initial list of doctrine items, representing all special knowledge that seems to be needed to solve the target problems.
- For each target problem, step through the user actions that lead to a solution. If the user has insufficient reason to take an action in the sequence of required actions based on the current doctrine and the problem statement, then add or modify an item of doctrine.
- Step through the solutions again, until a complete and unambiguous set of doctrine is ensured.

Part of the doctrine for OPSTrains is shown in Table 1. Each of the three designs required between 20 and 30 items of doctrine roughly the size of the items shown. Having completed the doctrine, we then evaluated the facility and expressiveness of the designs by doing programming walkthroughs on problems that had not considered in the design process.

OPSTrains, predictably, proved to be the most expressive. Perhaps surprisingly, it also showed the greatest facility, because the central ideas of the problem statement could be translated quite directly into OPSTrains rules guided by doctrine of about the same size and complexity as for the other designs. ZeroTrains placed last in both dimensions, because each solution required work involved in overcoming the lack of power. ShowTrains also had many of these shortcomings.

Table 1. Excerpts from the OPSTrains doctrine.

RO1:	If	starting,
	then	draw a picture of envisioned interface as it would initially appear to the user.
RO2:	If	an object can move or can be placed in a particular area of the interface that is not drawn,
	then	draw an object that is big enough to contain the objects, and specify that the object is to be "Hidden in Simulation."
RO3:	If	an object can move or be placed on top of an object that is too small to contain the object,
	then	draw an object that is big enough to contain the object, and specify that the object is to be "Hidden in Simulation."
...		
RR5:	If	an object in the pattern of a rule may match any object regardless of its display,
	then	specify that this object is a variable. (a big V will be placed over the variable object in the pattern)

III. EMPIRICAL RESULTS – THE CHEMTRAINS PROTOCOLS

The walkthrough results clearly indicated that OPSTrains was the design of choice, both in expressiveness and facility. But did our walkthrough analysis reflect reality? It is possible that our assessment of the decisions faced by users, or the ability of the OPSTrains doctrine to guide these decisions, were very wrong.

The ChemTrains Prototype

We implemented a prototype of ChemTrains based on the OPSTrains design. A graphical simulation program in this prototype is built by first drawing a picture of the simulation as an end user would initially see it. The user specifies changes that should occur on the simulation by drawing pattern/result rules. A rule editor allows the user to draw a pattern picture that is to match a subset of graphical objects on the simulation, and a result picture that is to replace these matched graphical objects. Figure 1 shows the prototype, which contains the main simulation picture, the pattern and result pictures of a single rule, and the commands for editing these pictures. The example shown in this figure is one of our original six problems, the Bunsen Burner problem, stated in Table 2. The user tests the simulation by entering "Execute Mode." In this mode, when the pattern of a rule matches objects in the simulation, ChemTrains fires the rule and replaces the matched objects with the objects in the result picture of the rule. The rules continue to fire until no rules match. If the user changes the screen during execute mode, the rules are again executed.

Table 2. The bunsen burner problem.

Show how the flame of a bunsen burner responds when the user moves a control knob to the off, low, or high position. Show water in a beaker above the flame changing from ice to water to steam as the flame changes.

Our claim from the programming walkthroughs was that a nonprogrammer could use this version of ChemTrains to solve the problems we had considered, relying only on the knowledge described by the doctrine. To test this claim, we asked subjects to program the bunsen burner simulation, using the prototype.

Subjects and Method

Six subjects performed the task, working individually. Subjects were all volunteers, with five having backgrounds in anthropology, chemistry, or psychology. Of these five, four had some traditional programming experience. A sixth, added as an informal control, was an experienced programmer. All had completed at least a four-year university degree. All protocols were videotaped, and subjects were asked to think aloud as they worked. [8]

The instructional material given to the subjects was essentially the same doctrine used in the programming walkthroughs. We rewrote the doctrine in complete sentences and made some word substitutions to avoid jargon (see Table 3 for the rewrite of the original doctrine shown in Table 1). The first three subjects were given the doctrine in written form and asked to read through it. For the second three, an experimenter went through the doctrine with the subject, briefly explaining it and making pencil sketches to illustrate most items. The subjects were then given a written statement of the bunsen burner problem (Table 2) and asked to program a solution using the ChemTrains prototype. Because we were interested in the sufficiency of the doctrine, not in a subject's ability to memorize it, the written doctrine remained available to subjects as they worked. Three of the subjects operated the ChemTrains prototype directly; the other three gave instructions to an experimenter, such as "draw a box on the screen," and the experimenter operated the interface.

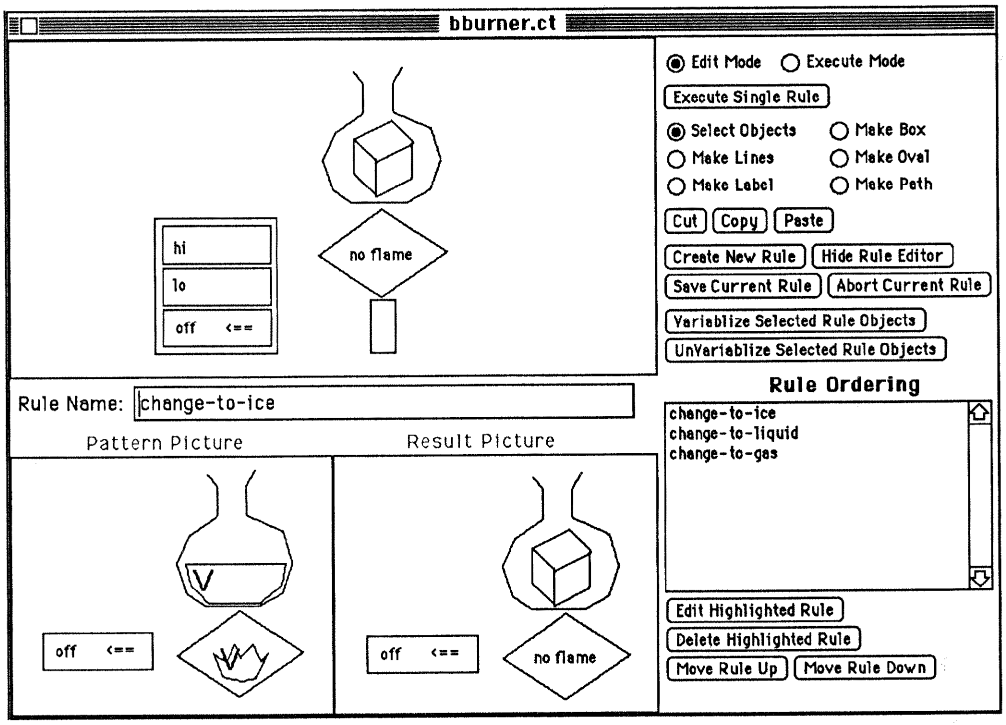

Figure 1. One possible solution to the bunsen burner problem. The main window (top) shows a beaker containing an ice cube. Below it is a diamond-shaped holder object for the flame. The control box contains another holder that surrounds the "off" and "<==" labels. In the pattern picture, the flame and the water have been variabilized.

Table 3. Excerpts from the doctrine as used by the empirical subjects (compare Table 1).

To begin, draw a picture showing the things you want to see on the screen initially.

Things can be inside other things.

You can draw things wherever you want, but rules can't put things in the middle of nowhere on the screen. If you need a thing to appear somewhere where there is no thing to put it inside of, you can create a new thing to act as a holder. If you don't want to see the holder thing you can make it invisible when the program runs.

If you want a thing to appear near a thing that is too small to hold it, create a new thing big enough to hold both. You can make the holder invisible when the program runs.

...

If a thing in the Pattern picture has to be there but could be any kind of thing, indicate that it is a variable. It will be marked with a big V.

We established several levels of fallback for subjects who encountered difficulty with the task. First, after letting them struggle for a brief time, we directed them to review the doctrine. Second, if this general review wasn't helpful, we pointed out the specific item of doctrine that would help them with their current problem. Third, if they were still lost, we explained how the conditions for the given item of doctrine were matched by the current situation, without explaining what action the doctrine implied. Finally, if this failed or if no item of doctrine addressed the problem, we explained the solution.

Results

All six subjects were able to apply the more straightforward items of doctrine and make initial progress toward a solution. The experienced programmer, our first subject, completed the entire task with only minor problems. The next two subjects found the doctrine inscrutable and the graphics interface frustrating; it was at this point that we decided to provide a better explanation of the doctrine and to have the experimenter operate the interface. Note that we were interested in the sufficiency of the doctrine, not the quality of its write-up or the ease with which the subjects could learn and use the interface.

Even with the improved explanation of doctrine, there were several areas in which subjects had difficulties. Major areas of success and difficulty for all subjects are summarized in Table 4.

Table 4. Important successes (shown as '+') and problems (shown as '-') of subjects using ChemTrains.

Success or problem (before fallback)	*number of subjects. (out of 6)*
General doctrine	
+ started by drawing picture of bunsen burner on screen	6
+ created rules with "pattern" and "result" part	6
+ copied main picture into pattern and result	5
+ modified result to show desired changes	5
Overall program operation	
- didn't understand how user would interact with simulation	6
- confused about how rules affected the screen at run-time	6
- thought rules fired only when user clicks on rule name	1
- thought different screen needed for each result	1
- thought pattern should contain burner control, result should contain burner itself	2
- thought control and flame should be off in pattern on in result	2
Holders and adjacency	
- didn't realize need for holders	6
- expected system to distinguish adjacency relationships	6
- assumed flame would be drawn above burner	3
- assumed positioning control knob next to label would work	5
- assumed steam would be drawn above beaker	1
Variables	
+ used variables correctly without prompting	1
- needed variables but didn't think of using them	4
+ recognized that large number of rules was a potential problem	2
+ recognized need to show generic pattern	1
- identified how to use variables correctly, but tried to understand how they worked before applying them	3
- made holder a variable instead of its contents	1
- considered making the control a variable -- "it varies"	2
- thought system would match all possibilities as a variable	1
- thought system would match empty space like a variable	1
Multiple objects (subject used more than one ChemTrains object to represent a single real-world object or substance, and assumed ChemTrains would treat the multiple objects as one.)	
- treated label and box as a single object	1
- treated steam bubbles as a single object	2
- treated lines of ice as single object	1

As Table 4 indicates, all subjects had some problem understanding the general way in which ChemTrains operated. Most of this seemed to clear up as soon as subjects completed their first rule and saw it execute. Even though the doctrine explained how to use holders, all subjects assumed that ChemTrains would recognize when two objects were close to each other, an assumption that blocked their ability to recognize places where holders were needed. This was also a fairly easy misconception to correct. Except for the experienced programmer, subjects had the greatest difficulty applying the section of the doctrine describing the use of variables. Variables are not essential to the bunsen burner solution, but they allow it to be programmed with fewer rules, since a variable can match (for example) either water or ice in the beaker, transforming either to steam when the control is placed on high. Even after fallback hints helped subjects identify the doctrine explaining how to use variables, they hesitated to apply the doctrine, expressing uncertainty as to what should be variabilized or whether variables were the right solution.

IV. DISCUSSION – EMPIRICAL FINDINGS COMPARED TO WALKTHROUGH PREDICTIONS

At the outset we note that the help provided through fallbacks placed all subjects in a better position than a naive nonprogrammer who had simply read the doctrine (cf. Perkins, [14] for the beneficial effect of similar fallbacks). This paper examines problems that evidently go beyond simple accessibility of knowledge.

We place subjects' problems into three categories, each of which has implications for the programming walkthrough methodology and for ChemTrains. First, some of the problems were caused by *holes in the doctrine*. Subjects' inability to comprehend how a user would interact with the system and how the firing of a rule would affect the main screen fall into this category. In writing the doctrine we were simply too close to the problem to appreciate the need to state all the basic facts. We might have avoided this failure by performing the walkthrough at a more detailed level, demanding a chain of exact or near matches leading from the task description through elements of doctrine to a solution, in the manner of Kintsch's construction-integration model of text comprehension [7]. However, the implications for the ChemTrains design are minimal, since the missing knowledge was easily comprehended by subjects when it was presented to them.

A second category of problems occurred when *subjects' real-world knowledge interfered with doctrinal prescriptions*. This was the case with holders and adjacency. The relevant item of doctrine was fairly clear as to where holders were required, and subjects understood the doctrine once it was pointed out to them in context. But every subject initially assumed that ChemTrains could distinguish a pattern by evaluating how close two objects were to each other, just as there is a real-world distinction between a knob pointing at "on" and one pointing at "off," or a pot on the stove and one on the table. Interference from real-world knowledge also explains the difficulties subjects had when multiple graphical objects represented a single real-world object.

Predicting interference from real-world knowledge during the walkthrough is difficult, and adjusting the doctrine to trap all possible misconceptions would be impossible: the doctrine simply can't be extended to describe everything that the language cannot do. This may be an area where empirical testing is required to identify particularly attractive false paths that the doctrine should block. For the ChemTrains design, revised doctrine is a possibility, but we are considering adding the ability to recognize adjacency relationships, which all subjects expected, as well as the ability to group multiple objects so they will act as one.

The third category of problems involves situations for which the doctrine prescribed programming procedures, but *the doctrine didn't convince the subject that the recommended procedures would work*. The notable example here is variables. One of us (CL) has advocated eliminating variables from the ChemTrains design since its initial conception, because they make it difficult to read and understand an existing program. But program readability was not directly addressed by either the walkthrough or the empirical test. Because the doctrine stated exactly where and how a variable should be used, the walkthrough predicted that a programmer familiar with the doctrine should be able to write programs using variables. To use Soloway's terminology, the doctrine supplied the necessary plan.

Subjects' initial fumbling with the concept of variables indicated that the plan was indeed sufficient, since four subjects, including the experienced programmer, were able to read the item of doctrine and correctly state which object in the rule should be variabilized. A fifth subject realized that an object should be variabilized, but initially targeted the wrong object. However, none of the nonprogrammer subjects was comfortable with this approach. Having determined how to apply the doctrine, they tried to *understand* how it would affect the program, and they tried to identify other items of doctrine that might yield better solutions to their current problem. These efforts actually prevented them from acting on their original, correct interpretation of the doctrine. A similar desire for understanding was evident when subjects learned about holders. Thus doctrine that is informationally adequate may fail because users do not feel that they understand it.

The behavior we observed is similar to the metacognitive strategies of math students reported by Schoenfeld [15]. Faced with several potential paths to a geometry proof, Schoenfeld's subjects used rough pencil drawings to convince themselves that a proof would succeed before committing to the work of a formal construction with compass and ruler. Carry, Lewis, and Bernard [4] analyzed protocols of novice algebra students trying to solve equations. Like our empirical subjects, and unlike the oversimplified situation envisioned by the walkthrough, these students were often uncertain as to which of several operations to apply, and they used several strategies to guide the selection process. The need for such metacognitive strategies has often been claimed (see review in Alexander and Judy [1]).

Recent work by Anderson [2] suggests that there is good reason for our subjects' cautious strategy. Faced with potentially infinite combinations of operations that might solve a problem, the rational problem solver must choose, based on prior experience, an approach that advances toward the solution at an acceptable cost. In attempting to apply this strategy, the problem-solver will find it difficult to select among items of doctrine that prescribe actions but fail to describe their results. We conjecture that this difficulty is not so great in the case of doctrine such as "draw a picture," since subjects have prior knowledge of at least the surface effects of drawing a picture -- effects which partially match the goal of producing an animated on-screen graphic.

This analysis implies that doctrine should describe the results of each action it prescribes, so the programmer can confidently select and take the action when applicable. Describing the effect of variables raises again the issue of their implications for program readability, and we conclude that variables must be reconsidered for the ChemTrains design.

V. SUMMARY

The programming walkthrough is a knowledge-level approach to evaluating the power and ease of use of a programming language design. We judge the method to be useful but not infallible. Using design and doctrine vetted by a walkthrough, six subjects were all able to make significant progress in solving a well-defined task. The difficulties that subjects had could apparently be corrected with minimal changes to the design or the doctrine. However, the doctrine suffered from hidden assumptions and failure to counter preconceptions based on real-world knowledge, both of which the walkthrough failed to identify. The walkthrough was also unable to predict the effect of metacognitive strategies, which may require knowledge beyond what a knowledge-level analysis of the task would suggest.

ACKNOWLEDGEMENTS

We thank Nancy Pennington and Ray MacCall for suggestions concerning this work, and Jonathan Bein and the anonymous reviewers for comments on this paper. Thanks also to our six volunteer subjects for their time, effort and suggestions. This work was supported in part by grant no. IRI-8944178 from the National Science Foundation. Related research was supported by US West Advanced Technologies.

REFERENCES

1. Alexander, P. and Judy, J. The Interaction of Domain Specific and Strategic Knowledge in Academic Performance. *Review of Educational Research 58*, 1988, 375-404.
2. Anderson, J.R. *The Adaptive Character of Thought.* Hillsdale, NJ: Lawrence Erlbaum Associates, 1990.
3. Brooks, R. Towards a Theory of the Cognitive Processes in Computer Programming. *International Journal of Man-Machine Studies 9*, 1977, 737-751.
4. Carry, L.R., Lewis, C., & Bernard, J.E. Psychology of Equation Solving. Technical Report. Department of Curriculum & Instruction, The University of Texas at Austin, no date.
5. Forgy, C.L. OPS5 User's Manual. Technical Report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, July, 1984.
6. Green, T.R.G, Bellamy, R.K.E, and Parker, J.M. Parsing and gnisrap: A model of device use. In G. Olson, S. Sheppard, and E. Soloway (Eds.) *Empirical studies of programmers: Second workshop.* 1987, Norwood, NJ: Ablex, 132-146.
7. Kintsch, W. The role of knowledge in discourse comprehension: a construction-integration model. *Psychological Review* 95, 2, 1988, 163-182.
8. Lewis, C. Using the 'thinking-aloud' method in cognitive interface design, IBM Research Report RC 9265 (#40713), 1982.
9. Lewis, C., Polson, P., Rieman, J. and Wharton, C. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In *Proceedings of CHI'90.* ACM, New York, 1990, 235-242.
10. Lewis, C., Rieman, J. and Bell, B. Problem-Centered Design for Expressiveness and Facility in a Graphical Programming System. Technical Report CUCS-479-90, Department of Computer Science, University of Colorado, July, 1990.
11. MacLennan,B. *Principles of Programming Languages.* New York: Holt, Rinehart and Winston, 1987.
12. Maulsby, D. and Witten, I. Inducing programs in a direct-manipulation environment. In *Proceedings of CHI'89.* ACM, New York, 1989, 57-62.
13. Newell, A. The Knowledge Level. *AI Magazine.* Summer 1981. 1-20.
14. Perkins, D.N., and Martin, F. Fragile Knowledge and Neglected Strategies in Novice Programmers. In E. Soloway & S. Iyenger (eds.), *Empirical Studies of Programmers.* Ablex Publishing, Norwood, NJ, 1986, 213-229.
15. Schoenfeld, A.H. Beyond the Purely Cognitive: Belief Systems, Social Cognitions, and Metacognitions as Driving Forces in Intellectual Performance. *Cognitive Science*, 7, 1983, 329-363.
16. Sime, M.E., Green, T.R.G., and Guest, D.J. Scope marking in computer conditionals- A psychological evaluation. *International Journal of Man-Machine Studies*, 9, 1977, 107-118.
17. Soloway, E. and Ehrlich, K. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SWE-10, 1984, 595-609.
18. Spohrer, J., Soloway, E. and Pope, E. A goal/plan analysis of buggy Pascal programs. *Human Computer Interaction*, 1, 1985, 163-207.